FP7-ICT-217069

SEPIA

Secure Embedded Platform with Advanced Process Isolation and Anonymity capabilities

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Security Enhanced Mobile Platform Design

(Deliverable D1.3)

Due date of deliverable: May 31, 2011

Actual Delivery date: May 31, 2011

Start date of project: 01.08.2010

Duration: 30 months

Organisation name of lead contractor for this deliverable: ARM Ltd

Revision: 0.5

Project co-funded by the European Commission within the 7th Framework Programme (2010-2013)		
Dissemination Level		
PU	Public	☒
PP	Restricted to other programme participants (including the Commission Services)	☐
RE	Restricted to a group specified by the consortium (including the Commission Services)	☐
CO	Confidential, only for members of the consortium (including the Commission Services)	☐

Notices

This document is intended to fulfil the obligations of the SEPIA project concerning deliverable D1.3, described in contract number 217069.

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Table of Revisions

Revision	Date	Description and Reason	By	Affected Sections
0.1	04.12.2011	First full draft	Gregory Conti	All
0.2	04.28.2011	Second draft, added ARM inputs in chapter3	Gregory Conti	All
0.3	05.4.2011	third draft, TUG review and ARM corrections	Gregory Conti Kurt Dietrich	All
0.4	05.10.2011	G&D added chapter 3.1.2,3.2.2,3.2.3 & 4.1 TUG added chapter 3.2.4 & 4.2	Franz Hauswirth Marc-Michael Bergfeld Kurt Dietrich	3 & 4
0.5	05.25.2011	Update thanks to BrightSight review (1 & 3.7). Update of 3.6 thanks to TUG inputs	Gregory Conti	1 & 3.7 & 3.6
1.0	03 Jun 11	Review by coordinator	Roderick Bloem	none

Authors

Gregory Conti (ARM)

Franz Hauswirth (G&D)

Kurt Dietrich (TUG)

Dr. Stephan Spitz (G&D)

Prof. Dr. Marc-Michael Bergfeld (G&D)

Jan Blonk (BrightSight)

Executive Summary

This document explains the concepts of SoC hardware and software virtualization that provide strong isolation between a Trusted World and a Non-Trusted World.
In this document, the ARM TrustZone Technology is extended at SoC level to create a Trusted Generic System Architecture. The concepts of which are explained from the hardware architecture level up to the Trusted OS software architecture level in accordance with the requirements listed in the Sepia Mobile Platform Attack and Thread Models document and be OMTP TR1 Profile2 compliant.
To do so, Sepia partners by taking existing knowledge and combining it in a new systemic manner describe in this document new architecture and design principles such as Trusted Touchscreen Display, SoC certificate infrastructure standardization based on X.509 certificates and NSA suite B cryptographic key length, anti-cloning protection, trustworthiness proving and state of the art trusted operating system architecture.

Purpose

A fundamental issue of mobile platforms is how to guarantee sufficiently strong isolation between software components belonging to different stakeholders. Virtualisation techniques try to tackle this problem by partitioning a platform into strongly isolated islands. Any communication between these islands is typically subject to restrictions enforced by some kind of monitor, hypervisor or microkernel. Inside these partitions, managed environments like Java or .NET/CLR can be employed to allow fine-grained subdivision down to class, function or instruction level.

The objective of this task is to create a security enhanced virtualization environment for use in the SEPIA platform. In this task our focus will be on selecting and enhancing existing mobile virtualization solutions to allow seamless integration with the SEPIA platform hardware and software

Intended Audience

This document is intended to explain to the audience the hardware and software security architecture concepts for strong isolation of a Trusted World and a Non-Trusted World at SoC level. These concepts are the foundations for the Sepia Platform of WP4 research project and aims to align the research partner on the TrustZone Technology Based Generic System Architecture.

Background

ARM Ltd has developed the TrustZone Technology and the TrustZone Technology Based Generic System Architecture.

Giesecke & Devrient has developed the secure execution environment MobiCore for mobile devices based on the TrustZone Technology of ARM.

Contents

Table of Figures

Table of Tables

Glossary

DSA: Digital Signature Algorithm
ECC: Elliptic Curve Cryptography
HUK: Hardware Unique Key
MMU: Memory Management Unit
MPU: Memory Protection Unit
NV: Non volatile
PKCS: Public Key Cryptography Standard
PSS: Probabilistic Signature Scheme
RSA: Rivest Shamir Adleman
SoC: System on chip
SCP: System Control and Power sub-system
TEE: Trusted Execution Environment
UEFI: Unified Extensible Firmware Interface

1 Introduction

Embedded devices are becoming increasingly complex. Many now provide a high level of computational power and connectivity to the internet. They are also increasingly using open software platforms that permit users to download third-party applications. This, however, increases the security risk and is in conflict with these devices being used to handle data of increasing value, for example, banking credentials and user private informations.

Embedded devices must provide a way of protecting these data from malicious attacks. Software only security solutions as seen on PC are heavily criticized for not being able to protect as wished user privacy and e-business in our day life. They are also criticized for their high power consumption due to their security strategy mainly based on anti-virus. In order to change this statement of fact, ARM has developed the TrustZone technology which provides hardware security robustness by creating a virtual Trusted World protected from the pollution of Rich OS malwares (Non-Trusted World). The TrustZone technology is at the heart of each 'A' Class Cortex CPU already integrated into hundreds of millions of smart phones, tablet computers and other platforms on the market. But, the CPU is not the only actor participating into the system security of a SoC. As matter of fact, keyboard, display, touchscreen, but also system functions such as boot, debug, test manufacturing are also key components participating into the overall security scheme. In the same way that ARM has created hardware isolation between the Trusted and Non-Trusted world at CPU level; there is a need today to extend this isolation to all the SoC key components. This is the primary goal of SEPIA partner's research in this document. The isolation extensions at SoC level need to be made in such a way that high performance and low power capability are conserved. By doing so, the results of this research indicate what security sensitive aspects of the SoC can be seamlessly moved from the Non-Trusted World into the Trusted World. In this research, MobiCore from G&D is the highly secure operating system running in the Trusted World that is hardware isolated from the Rich OS.

This document primarily explains new architecture and design principles for SoC system level hardware and software virtualization that provide strong isolation between a Trusted World and a Non-Trusted World but also between Trusted applications run by MobiCore in the Trusted World.

This document also introduce in this context new architecture and design principles such as Trusted Touchscreen Display, SoC boot certificate infrastructure standardization based on X.509 certificates and NSA suite B cryptographic key length, anti-cloning protection, trustworthiness proving.

2 ARM SoC Hardware virtualization architecture for TEE isolation

2.1 ARM TrustZone technology introduction

ARM TrustZone technology provides a methodology that can be used to build a robust security solution. The basic principle behind TrustZone technology is the isolation of all software and hardware states and resources into two worlds:

- Trusted world. All high value assets, including code, data and hardware states that need to be guarded against malicious attack are placed in this world. Accesses to these assets are only permitted for software or hardware processes that are trusted.
- Non-trusted world. All other code, data and states that is not in the trusted world, resides here. In this world, from a user point of view, the assets are deemed to either:
    - have a low security value, or
    - when exploited, will not cause any high value assets to become visible or vulnerable to attack.
    - Accesses to these assets are permitted for software or hardware processes that are both trusted and non-trusted.

What constitutes as high value assets varies with application, but typically they are:

- The data/methods that require protection. For example:
    - hardware/software keys, certificates and licenses,
    - encryption/decryption algorithms or engines,
    - sensitive business logic such as risk management profiles,
    - high value media protection mechanisms.
- System states that can be exploited to allow access to the above. For example, memory maps, memory management settings, power and clock control, debug control and test manufacturing control.

TrustZone technology ensures that there is a strict isolation between these two worlds, and switching between trusted and non-trusted world is strictly controlled.
To differentiate between assets in trusted world and in non-trusted world, TrustZone technology defines two states that are used to mark all processes within hardware and software. These two states are:

- Secure, if they reside in the trusted world, or,
- Non-Secure, if they reside in the non-trusted world.

Note that in the industry, especially in organizations that make use of tamper resistant smart cards, the term “Secure” refers to a level of security that is far higher than the security claimed in this document, and often also require the addition of physical protection. The security goal of this system design is to define a trusted execution and storage environment which is far more secure than the general purpose “rich” operating

system of the device, and yet more flexible and integrated than the higher security environment provided by a Secure element such as a smart card. In the scope of this document, the terms Secure and Non-Secure are used to define states as used in ARM architecture definitions.

TrustZone technology is closely integrated into ARM processor architecture. It is also implemented in many other ARM IP components. These include debug subsystems and a variety of system components. This pervasiveness of security down to the actual components holding the assets is what makes TrustZone technology distinct from generic CPU based security. For more details on TrustZone system design methodology and on IPs that implements TrustZone technology, please see the following document.

- *ARM Security Technology* [1]. This document provides a general guide to issues around security in an embedded system, and on how to make a system more secure with TrustZone technology. It also covers both hardware and software aspect of a system when using TrustZone technology.

In later sections, we then look into how SoC system hardware must be minimally architected in order to provide a trusted system that is targeted towards the use of trusted operating systems.

2.2 ARM TrustZone CPU virtualization mechanism

All ARMv7-A processors implementing the ARM instruction set also implement architectural security extensions. The key functionalities are

- Two main core states are implemented (conceptually, two virtual cores), having a Secure and Non-Secure state. When the processor is in Secure state, it is operating in the trusted world, and conversely, when the processor is in Non-Secure state, it is operating in the non-trusted world. The combination of mode and ‘NS-bit’ in the Secure Configuration Register (SCR) indicates which state the processor core is currently in:
    - when NS-bit = 0 or when the processor is in monitor mode, the core is in Secure state. (Not in Non-secure state)
    - and when NS-bit = 1 and not in monitor mode, the core is in Non-secure state.
- A Monitor mode is also defined and is designed to provide a bridge between the code running in the Non-Secure state and Secure state. When in Monitor mode, the processor is in the Secure state regardless of the NS-bit state. The software running in the monitor mode is primarily used to perform the following before switching world:
    - save the state of the current world
    - restore the state of the world to switch into
- A Secure Monitor Call (SMC) exception instruction is implemented, which is used by software to initiate a switch to monitor mode.
- Interrupts and exceptions can also be configured to cause the processor to switch into Monitor mode, Secure state or Non-secure state. Independent exception vector tables are implemented for each state, and for the Monitor mode.

- Switching a processor core to Monitor Mode (using any of the above techniques) causes execution to vector to predefined location, and hence places further execution under control of trusted code at that location.
- For Memory Management, different translation tables are used for accesses originating from the trusted world, and accesses originating in the non-trusted world.
    - The tables for accesses originating from the trusted world are held in memory belonging to the trusted world, and provide an additional bit in the translation tables that allow such accesses to be made to memory belonging to either the trusted world or the non-trusted world.
    - The tables for accesses originating from the non-trusted world can only access memory belonging to the non-trusted world.
    - The security designation of the translation tables is maintained within the Translation Lookaside Buffer (TLB) so that TLB lookups by accesses originating from the trusted world will only access TLB entries derived from Secure translation tables, while TLB lookups by accesses originating from the non-trusted world will only access TLB entries derived from the Non-Secure translation tables.
    - The memory management unit (MMU) also implements the capability to dedicate memory address spaces to privilege or un-privileged software in conjunction with the Execute Never mechanism for protecting against many buffer overflow and privilege escalation types of attacks.
- The tags of each cache line will also record the NS bit which indicates which world the cache line maps to.
- Bus Interfaces for main memory or system access will also implement flags that define the world that an access belongs to. For example, for an AXI access, the NS state is defined using the ARPROT[1] and AWPROT[1] signals, which indicate the secure state of each access.

For more details please refer to the following:

- *ARMv7-A Architecture Reference Manual* [2]. This is ARM's Application class Architecture Reference Manual, and includes details on how TrustZone technology is implemented in its architecture.

2.3 ARM SoC System resources virtualization

The key to implementing security is extending the “two worlds” isolation beyond the CPU. This is done using the Non-Secure (NS) flag with bus accesses in the system. For example, on an AXI bus, the NS flag is mapped to ARPROT[1] and AWPROT[1]:

- AWPROT[1]: Write transaction – LOW is Secure and HIGH is Non-secure.
- ARPROT[1]: Read transaction – LOW is Secure and HIGH is Non-secure.

All masters in the system will have to set these flags accordingly. Non-trusted world accesses must have these flags set to HIGH. Trusted world accesses must have these flags set to LOW.
The NS flag acts like an additional address bit to the address, creating two distinct address spaces:

- A Secure address space that belongs to the trusted world.
- A Non-Secure address space that belongs to the non-trusted world.

By hardware design, non-trusted world masters must be made unable to access the Secure address space and will only be allowed to generate non-trusted accesses. Trusted world masters can access the Secure or Non-Secure address spaces.
A slave device can be placed into the secure address space or the non-secure address space either by the device itself interpreting the NS flag, or by intermediary devices such as the interconnect matrix decoding the NS flag on behalf of the slave. When an access targets a device that is not mapped to the same world, the transaction will fail with either the slave device, or the intermediary on behalf of the slave, silently ignoring the access, or returning an error response.
For example, an APB slave peripheral can be placed in the trusted world by the AXI bus fabric decoding the target address and checking that the associated access signal AWPROT[1] or ARPROT[1] is set to LOW.

In concept the Secure address space and the Non-Secure address spaces are distinct, but it is permissible and common that a data location is aliased between the Secure address space and the Non-Secure address spaces. However, this approach is not recommended for executable memory and data location that might be cached in an implementation because this can lead to the standard coherency issues that arise from having different physical aliases of the same location. To prevent such issues, both address spaces must be strongly-ordered.
For memory-mapped peripherals, the approach can be used for the following:

- to allow a peripheral that is capable of being shared between the trusted world and the non-trusted world to determine whether a request has come from a master operating in the trusted world or from a master operating in the non-trusted world.
- to make a peripheral that is originally mapped to the Non-Secure address space also visible in the Secure address space. This is useful to allow secure masters that do not have the ability to generate Non-Secure accesses to access the Non-Secure peripheral.

The decision as to where a peripheral resides can be dynamically made by software. In that case, care has to be taken not to accidentally transfer data when transferring the mapping, and to only allow trusted world access to configure it.

TrustZone Technology Based Generic System Architecture

Figure 1 shows a high level system topology of a typical SoC that incorporates a Trusted Base System Architecture. This trusted base system architecture is built upon the hardware virtualization of SoC Initiators, memories, system functions and SoC peripherals.

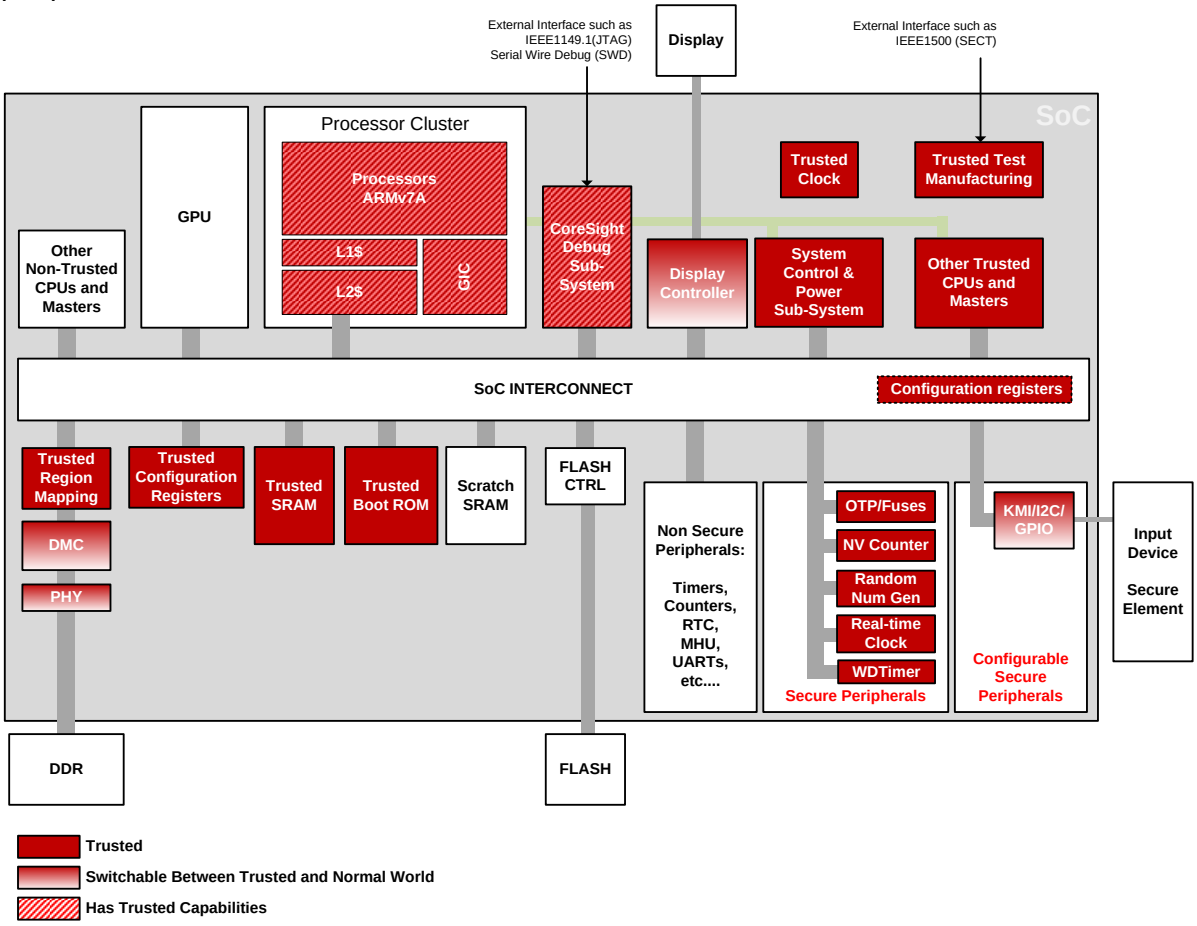

Figure 1: TrustZone Technology Based Generic System Architecture

The key modules related to security that implements hardware virtualization are:

- CPU and Initiators:
  - 'A' Class ARM processor(s) with TrustZone Technology
    - Generic Interrupt Controller (TrustZone Technology compliant)
  - Display Controller (TrustZone Technology compliant) capable of issuing secure accesses for fetching secure frames in trusted DDR after its controlling interface have been dedicated to the trusted world and controlled by MobiCore.
  - Trusted System Control and Power sub-system (SCP) which includes an 'M' Class ARM processor.
  - Other Trusted CPUs DMAs (TrustZone Technology compliant)

- Interconnect and memory resources:
    - SoC Interconnect with Trusted Configuration Registers
    - Trusted SRAM where MobiCore and Trusted applications are located
    - Trusted Boot Rom
    - Trusted Region Mapping Logic that allows dedicating some DDR data portion to the Trusted World (Trusted Frame buffer) and some other to the Non-Trusted World.
    - DRAM memory controller (DMC) that has trusted configuration capability
    - DDR Physical Interface (PHY) that has trusted configuration capability
    - Trusted One Time Program (OTP) ROM or Fuses
    - Trusted Non-Volatile (NV) Counter

- System functions virtualization:
    - Trusted System Configuration Registers
    - CoreSight Debug Subsystem (TrustZone Technology compliant)
    - Trusted Factory Test Manufacturing

- SoC peripherals virtualization:
    - Trusted Clock Source
    - Trusted Real-Time Clock
    - Trusted Watchdog Timer
    - Trusted Random Number Generator (RNG)
    - Trusted User Input Interface for connection of Trusted keyboard and Trusted mouse
    - Trusted User Input Interface such as I2C for connection a secure elements (RFID, SIM, NFC) and touchscreen controller.
    - Trusted General Purpose Input and output (GPIO)

Trusted Touchscreen Display capability that is newly introduced by Sepia partners' research is made by combining the Trusted Display capability with a Trusted I2C connection and usage of Trusted GPIOs. By doing so, low cost touchscreen controllers can be connected and securely controlled by MobiCore and allows providing for the first time on mobile devices and more generally on computers a complete trusted user interaction solution that is capable of running in parallel to an open and potentially unsafe operating system. The Sepia partners' research indicates that the hardware cost to enable this trusted user interaction solution is negligible since it does not require the addition of a security IC.

2.4 ARM SoC Debug virtualization

An SoC debug solution is usually used to provide visibility into the system so that system states can be monitored, and debugging of software can be performed. Debugging types generally fall into two categories:

- external debugging
- self-hosted debugging.

The debug subsystem in the TrustZone Technology Based Generic System Architecture maintains the TrustZone two world methodologies for the external debugging and the self-hosted debugging.

External Debugging

External debugging allows a user to gain visibility and perform debugging without the need to add debug code to the application, operating system or other system code expected to run normally on the system. ARM architecture processors provide dedicated support within the CPU for external debugging using a special debug state. At the SoC level, a debug subsystem is usually implemented which completes the debug subsystem, providing visibility of other system devices. This debug subsystem also allows off-chip external debugger devices to connect to the system via interfaces isolated from the main system interfaces. Examples of interfaces dedicated to debug includes IEEE 1149.1 JTAG, a low pin-count Serial Wire Debug (SWD) interface, and other types of parallel and serial trace ports.
Debug operations within external debugging falls into two categories, invasive and non-invasive:

- Invasive debug. This generally refers to operations that interfere with the operation of the CPU and system. For example:
    - Debug operations via an external JTAG interface that performs halting or stepping of a CPU. These directly interfere with the operation of the CPU,
    - Accessing system states via the main system fabric. This allows an external debugger to directly modify data that may be used by the CPU, directly interfering with its operation. It also indirectly interferes with the operation of the CPU by, for example, causing delays on the system bus.
- Non-invasive debug generally refers to operations where system visibility is provided with little or no impact to the system normal operation. An example is the use of trace functionality, to provide statistics on Instruction execution, or bus utilization. Trace is usually completely non-invasive. However, in some systems the trace system may be such that using trace perturbs timings to some extent; for example, where trace uses bandwidth on system interfaces.

A debug system in a system that implements TrustZone technology must consider that debug functionality can be enabled or disabled for the whole SoC. It also must ensure that the non-trusted and trusted worlds remain isolated, for example when the non-trusted world debug is enabled while the trusted world debug is disabled but wish to be executable. The external debugger can have access to states in the Trusted World only if it is deemed to be trustworthy, for example using an authenticated debug certificate unique per SoC.

ARM processor architecture defines control signals that allow the system integrator to control debug accessibility and implement debug scenarios at processors level and at SoC level. These controls are the:

- Debug Enable (DBGEN) input signal, which when asserted, enables invasive and non-invasive debug of Non-secure state. Note that when DBGEN is not asserted access to debug components is generally still permitted, but those components are disabled.
- Non-Invasive Debug Enable (NIDEN) input signal, which when asserted, enables non-invasive debug operations, such as trace, of Non-secure state. NIDEN can be asserted independently of DBGEN.
- Secure Privileged Invasive Debug Enable (SPIDEN) input signal, which, when asserted along with DBGEN, enables invasive and non-invasive debug of Secure state.
- Secure Privileged Non-Invasive Debug Enable (SPNIDEN) input signal, which, when asserted along with NIDEN, enables non-invasive debug of Secure state.

All the input signals above (DBGEN, NIDEN, SPIDEN and SPNIDEN) can be utilized to control the debug at SoC system level and creates the following debug scenarios;

- No debug
    - Non-Trusted World is not debug-able
    - Trusted World is not debug-able
- Public debug
    - Non-Trusted World is debug-able
    - Trusted World is not debug-able but can be executed
- Secure privilege debug
    - Non-Trusted World is debug-able
    - Trusted World is debug-able

In addition, the MobiCore Trusted OS can control the non-invasive debug of code executing in Secure unprivileged modes (SUNIDEN) when either DBGEN or NIDEN is asserted, even when SPIDEN and SPNIDEN are not asserted to create the following scenario:

- Secure user debug
    - Non-Trusted World is debug-able
    - Trusted World trusted user apps (selectable) are debug-able but trusted privilege kernel is not debug-able.

Self-hosted Debugging

Self-hosted debugging, also known as monitored mode debugging, requires additional code in the application, the operating system and/or other system software in order to gain visibility and perform software debugging. There is, therefore, usually an impact on the performance of the software or operating system.

However, this form of debugging can be performed with little or no supporting hardware on or off chip. As a result, it is pervasive in many platforms, often deployed as a low cost debugging solution of third-party application code in the final product.
While self-hosted debugging can be achieved with little or no additional supporting hardware, additional hardware on-chip can often be added to enhance the debug visibility and reduce the impact on performance. For example, the ARM architecture supports:

- counters for performance monitoring of processor hardware
- software access to much of the debug subsystem components, such as:
    - hardware breakpoints and watchpoints to generate software exceptions, allowing them to be used during self-hosted debugging
    - trace to support software profiling.

Any additional hardware that exist needs to maintain the “two worlds” methodology, ensuring that accesses from non-trusted world can only access and monitor states from within the non-trusted world.

The same DBGEN, SPIDEN, NIDEN and SPNIDEN control signals and SUNIDEN software controllable function apply for self-hosted debug (invasive) and performance monitoring/profiling (non-invasive) as they do for external debug.

In addition, the MobiCore Trusted OS can control the invasive debug of code executing in Secure unprivileged modes (SUIDEN) when either DBGEN or NIDEN is asserted, even when SPIDEN and SPNIDEN are not asserted. Consequently the trusted OS need to take care in using this feature only for tasks which the user is suitably authenticated to debug.

3 ARM SoC Boot virtualization architecture for TEE isolation

3.1 SoC Root-of-trust key infrastructure concept

The following sections describe the Sepia partners' standardization choices around NSA suite B cryptographic key length and algorithms. The cryptography algorithms referred to are well-understood industry standards. The choice of algorithms and security level does not radically affect the key certificate infrastructure and the SoC trusted boot sequence (not described in this document) but it gives a useful indication of the size of the certificate structures needed.

Sepia partners recommend to use NSA suite B approved security algorithms and particularly the SECRET level security standard for consumer market also referred as 128 bits security:

- AES128
- SHA256
- RSA2048 (RSA-PSS) or ECC256 (EC-DSA)
- Which drive the need of the OPT/Fuse Hardware Unique Key (HUK) and the root of trust public hash stored in OTP/Fuse is required to be SHA-256 bits

Sepia partners also recommend to use NSA suite B approved security algorithms and particularly the TOP SECRET level security standard for government market also referred as 256 bits security:

- AES256
- SHA512
- RSA4096 (RSA-PSS) or ECC521 (EC-DSA)
- Which drive the need of the OPT/Fuse Hardware Unique Key (HUK) and the root of trust public hash stored in OTP/Fuse is required to be SHA-512bits

The 256bits security level is not explained in this document.

The choice given to use RSA-PSS or EC-DSA is mainly made for patent related purpose, hardware cost, software certificate size and time required for authentication.

Each certificate used in this document has been chosen by Sepia partners' to be X.509 v3 compliant for standardization purpose.
PKCS #1 (RSA), FIPS 186-3, SECG and P1363 standards must be followed.

Hashes

A cryptographic hash function is a deterministic procedure that takes an arbitrary block of data and returns a fixed-size bit string. Any accidental or intentional update to the data changes the returned value. The encoded data is called the message, and the

returned value is called the message digest or hash. Usually the hash is much smaller than the message.
The SHA-2 hash function family used as required by NSA suite B (128bits security level) produces a secure message digest of 224 to 512 bits. In this document, the minimum digest used is 256 bits.
Hashes are used during signature generation and integrity verification of the firmware images.

Symmetric Keys

Symmetric-key cryptography algorithms use trivially related, often identical, cryptographic keys for both decrypting and encrypting a message.
For an *Advanced Encryption Standard* (AES) symmetric key, the shortest key length to use is 128 bits.

Symmetric key encryption is used rather than asymmetric key encryption because it is much less computationally intensive, for keys of equivalent strength.

Asymmetric Keys

Asymmetric key algorithms create a mathematically related key pair, a secret private key and a published public key. Use of these keys can protect the authenticity of a message by using the private key to create a digital signature of the message, which can be authenticated using the public key.

Asymmetric key pairs are used to authenticate certificates that are digitally signed.

Digital Signatures

A digital signature is a mathematical scheme to verify the authenticity of a digital message. A valid digital signature informs a recipient that the message was created by a known sender and was not altered in transit. A digital signature scheme typically provides:

- A signing algorithm that produces a signature, from a message and a private key.
- A signature verifying algorithm that either accepts or rejects the message's claim to authenticity. It takes a message, a public key and a signature as input.

The RSA-PSS signature scheme additionally signs a hash of the message instead of the message directly. This technique is almost always used with RSA because the amount of data that can be directly signed is proportional to the size of the keys, which is usually much smaller than the original message.
A digital signature produced using RSA-PSS is the same length as the RSA key modulus, and so signatures have the same length as keys, that is, at least 2048 bits.

The EC-DSA signature scheme signs a hash of the message instead of the message directly. The notable benefit of using EC-DSA over RSA-PSS is that the resulting signature size is much smaller. The 256 (224) bit EC-DSA key length is comparable to the 2048-bit RSA-PSS key length in terms of security. [According to NIST] and is the suggested key length for the coming years from 2011 [NIST].

X509 certificates

Each certificate used in this document is required to be X.509 v3 compliant to ensure cross compatibility between tools, issuers and platforms.

Figure 2 is an overview of the structure of X.509 v3 certificate with example of filling according to certificates explained in next chapters.

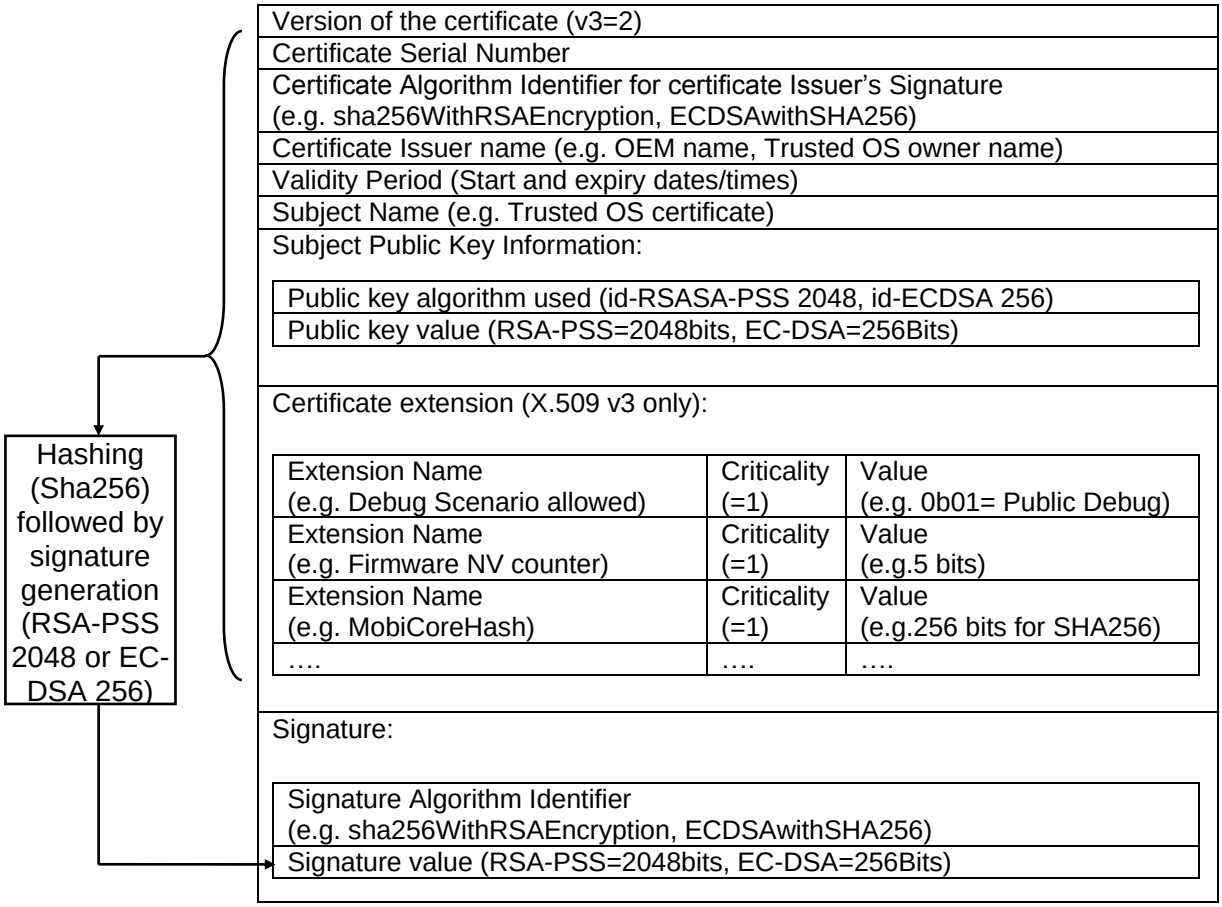

Figure 2: Structure of X.509 v3 certificate

OTP/Fuse Keys and Data

The following table shows the minimum data that the OTP/Fuse memory contains to be NSA suite B 128bits security level compliant.

Table 1 OPT/Fuse Data Fields

Name	Size (bits)	Description
Root of trust Public Key Hash	≥256	Message digest of the root of trust public key
Hardware Unique Key	≥128	Key used for Secure Storage, binding and key generation
Manufacturing Mode Control Bits	≥2	0b00: un-initialized device 0b01: Test and Development device 0b10: Mass production device 0b11: Device not working

The manufacturing control bits enable manufacturing mode or sub-features within this mode to be permanently disabled, before a device is shipped.

3.2 SoC certificates (owners and dependencies)

Key Infrastructure

Figure 3 explains the certificate key infrastructure in place for each certificate and how they are related.

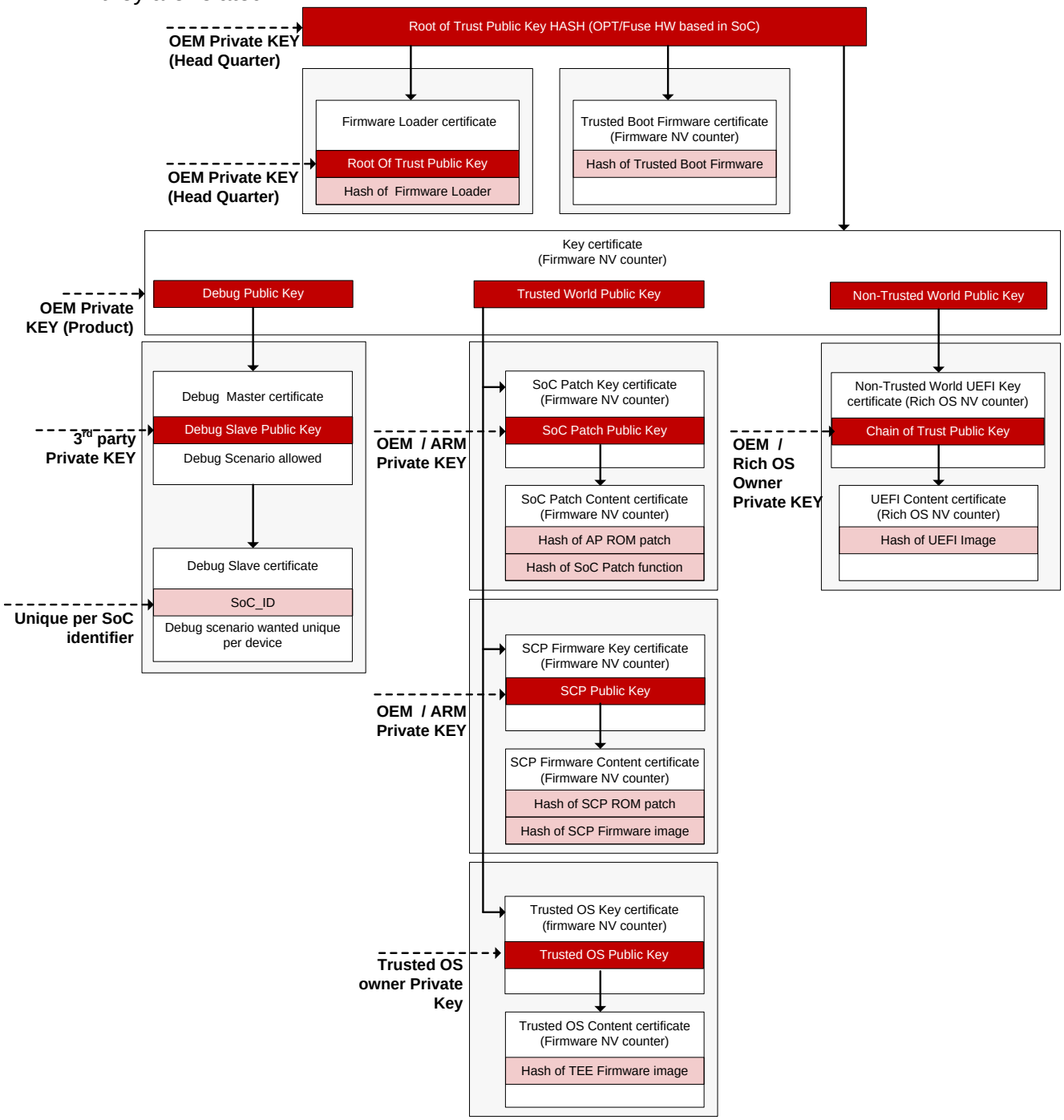

Figure 3: Certificate key infrastructure

The root of trust private key is the property of the OEM which is able to sign the subsidiary public keys (key certificate) owned by the OEM or third parties for which the

OEM or third parties own the corresponding private key and use them to sign the later firmware certificates.

Chain of Trust

Each step in the trusted boot process is only as trusted as the step that precedes it. Therefore a chain of trust is established.

Key security property

The first link in the chain of trust is the management of the root of trust private key, which is signing subsidiary keys used to sign all the subsequent certificates.
The root of trust private key is the property of the OEM which is able to sign the subsidiary public keys (key certificate) owned by the OEM or third parties, which OEM or third parties own the corresponding private key and use then to sign some parts of firmware.

If a single version of the root of trust private key exists for an entire class of devices then the security of all those devices is reliant on this key not being compromised. Although it might be difficult for an attacker to obtain these keys, the consequences of a successful attack are severe. This is known as a “class break”.

To mitigate such a “class break”, it is expected there are multiple root of trust private keys. Each key is shared by a smaller group of devices than the entire hardware platform. If a single key is compromised then only the group of devices that that key protects are compromised, not all devices for that hardware platform.
The question arises as to how many devices should each key protect? The cost of a successful attack to the device manufacturer is proportional to the number of devices that the attack compromises. Usually, it is too expensive to manage unique keys for every device but similarly the cost of a “class break” must be mitigated. There is a trade-off between the cost of a successful break and the cost of managing multiple keys. Each device manufacturer must decide where the balance lies.

To support multiple keys for a class of devices, a manufacturer must program the appropriate root of trust public key hash in each device’s OTP/Fuse memory, before it is shipped to customers.

If the third parties private keys are compromised they can be revoked, as long as the root of trust private is not compromised, by signing a new trusted key certificate with a NV firmware counter incremented.

The hardware unique key per device must be generated randomly per device to allow binding any secret uniquely to a platform in such a way that secrets in encrypted form cannot be exported to any other devices. Also compromising this key for one device should affect only this device if the security strategy of the Trusted OS is made accordingly.

Non-Trusted Firmware Loader certificate

The root of trust public key (OEMRootOfTrustedPK) is included in the Non-Trusted Boot Firmware certificate payload and is used to authenticate the Non-Trusted

Firmware Loader, Trusted Boot Firmware certificate and the Trusted Key certificate (the corresponding hash of it has been stored in OTP/Fuse during manufacturing time). Consequently, the root of trust public key (OEMRootOfTrustedPK) is hashed and compared with the SoC OTP/Fuse for verifying that the certificate has truly been issued by the OEM.
This certificate does not need to be protected against replay attacks.

Table 2 Non-Trusted Firmware Loader Certificate X.509 meaningful fields

Subject Name = **FirmwareLoaderCertificate**		
Certificate Serial Number		
Subject Public Key Information		
Certificate extension (X.509 v3 only):		
Extension Name OEMRootOfTrustPK	**Criticality** =1	**Value (2048Bits or 256Bits)** (RSA-PSS2048bits / EC-DSA256bits)
Extension Name FirmwareLoaderHash	**Criticality** =1	**Value (256Bits)** SHA256(Non-Trusted Firmware Loader Hash)

Trusted Boot Firmware Certificate

The Trusted Boot Firmware contains the software code that executes in the Trusted Execution Environment to perform the Trusted Boot process.

Table 3 Trusted Boot Firmware Certificate X.509 meaningful fields

Subject Name = **TrustedBootFirmwareCertificate**		
Certificate Serial Number		
Subject Public Key Information		
Certificate extension (X.509 v3 only):		
Extension Name FirmwareNVCounter	**Criticality** =1	**Value (5Bits)**
Extension Name TrustedBootFirmwareHash	**Criticality** =1	**Value (256Bits)** SHA256(Trusted Boot Firmware)

Trusted Key Certificate

The trusted key certificate is the first certificate to be downloaded during the Trusted boot process (not explained in this document) in the Trusted Execution Environment.

The root of trust public key (OEMRootOfTrustedPK) is included in the trusted key certificate payload and is used to authenticate the trusted key certificate (the corresponding hash of it has been stored in OTP/Fuse during manufacturing time and is only used to verify the issuer of the certificate). Consequently, the root of trust public key (OEMRootOfTrustedPK) is hashed and compared with the SoC OTP/Fuse for verifying that the certificate has truly been issued by the OEM.

The other public keys defined in the Trusted Key certificate are used to verify the later certificates in the chain of trust process. The corresponding private keys are still owned by the OEM but for revocation capability the Trusted Key Certificate acts as an authority transfer to OEM product teams.

Table 4 Trusted Key certificate X.509 extension meaningful fields

Subject Name = **KeyCertificate**		
Certificate Serial Number		
Subject Public Key Information		
Certificate extension (X.509 v3):		
Extension Name FirmwareNVCounter	**Criticality** =1	**Value (5Bits)**
Extension Name PrimaryDebugCertificatePK	**Criticality** =1	**Value (2048Bits or 256Bits)** (RSA-PSS2048bits / EC-DSA256bits)
Extension Name TrustedWorldPK	**Criticality** =1	**Value (2048Bits or 256Bits)** (RSA-PSS2048bits / EC-DSA256bits)
Extension Name NonTrustedWorldPK	**Criticality** =1	**Value (2048Bits or 256Bits)** (RSA-PSS2048bits / EC-DSA256bits)

Trusted Debug certificates

As described in the previous chapter the ARM TrustZone Technology allows enabling multiple debug scenarios at SoC level as describe in Table 5.

Table 5 Overview of the SoC debug scenarios

No Debug	**=0b00 default for mass production devices**
DBGEN	= 0 (Invasive debug enable)
SPIDEN	= 0 (Invasive Secure Privilege Debug Enable)
NIDEN	= 0 (Non-Invasive debug enable)
SPNIDEN	= 0 (Non-Invasive Secure Privilege debug Enable)
SUIDEN	= 0 (Invasive secure User debug enable)
SUNIDEN	= 0 (Non-Invasive Secure User debug Enable)
Public Debug	**=0b01**
DBGEN	= 1 (Invasive debug enable)
SPIDEN	= 0 (Invasive Secure Privilege Debug Enable)
NIDEN	= 1 (Non-Invasive debug enable)
SPNIDEN	= 0 (Non-Invasive Secure Privilege debug Enable)
SUIDEN	= 0 (Invasive secure User debug enable)
SUNIDEN	= 0 (Non-Invasive Secure User debug Enable)
Secure User Debug	**=0b10**
DBGEN	= 1 (Invasive debug enable)
SPIDEN	= 0 (Invasive Secure Privilege Debug Enable)
NIDEN	= 1 (Non-Invasive debug enable)
SPNIDEN	= 0 (Non-Invasive Secure Privilege debug Enable)
SUIDEN	= 1 (Invasive secure User debug enable)
SUNIDEN	= 1 (Non-Invasive Secure User debug Enable)
Secure Privilege Debug	**=0b11 default for Test and Development devices**
DBGEN	= 1 (Invasive debug enable)
SPIDEN	= 1 (Invasive Secure Privilege Debug Enable)
NIDEN	= 1 (Non-Invasive debug enable)
SPNIDEN	= 1 (Non-Invasive Secure Privilege debug Enable)
SUIDEN	= 1 (Invasive secure User debug enable)
SUNIDEN	= 1 (Non-Invasive Secure User debug Enable)

These scenarios can be configured though the chaining of two certificates;

- Primary debug certificate which is signed for a class of device
- Secondary debug certificate which is unique per device

The primary certificate is owned by the OEM and its purpose is to specify what scenarios are allowed to be configured by the secondary certificate that is signed by a third party. It contains a spare field of 64 bits and the public key needed to verify the secondary debug certificate (SecondaryDebugCertPK).

As an example, the OEM could sign a primary debug certificate which allows only the enabling of the public debug and some spare bits (assigned to modem debug for example) on a class of device without knowing which device. The third party could then select accordingly to its need which particular device is public debug-able (e.g. development) or not debug-able (beta testing).

Another example would be to have the OEM signing a primary debug certificate that allows enabling the secure privilege debug to the Trusted OS owner which in turn can allow secure user debug only to its third party. This processing can be done for the whole lifecycle of the device without the OEM intervention.

Table 6 Primary debug certificate X.509 v3 extension fields

<table>
<tr><td colspan="3">Subject Name = PrimaryDebugCertificate</td></tr>
<tr><td colspan="3">Certificate Serial Number</td></tr>
<tr><td colspan="3">Subject Public Key Information</td></tr>
<tr><td colspan="3">Certificate extension (X.509 v3):
<table>
<tr><td>Extension Name
DebugScenarioAllowed</td><td>Criticality
=1</td><td>Value
0b00= No Debug
0b01= Public Debug
0b10= Secure User Debug
0b11= Secure Privilege Debug</td></tr>
<tr><td>Extension Name
SPARE</td><td>Criticality
=1</td><td>Value (64bits)
64bits</td></tr>
<tr><td>Extension Name
SecondaryDebugCertPK</td><td>Criticality
=1</td><td>Value (2048Bits or 256Bits)
(RSA-PSS2048bits / EC-DSA256bits)</td></tr>
</table>
</td></tr>
</table>

The secondary debug certificate purpose is to select which debug scenario to enable on the platform and to specify the unique identifier per device (SOC_ID).

Table 7 Secondary debug certificate X.509 v3 extension fields

Subject Name = **SecondaryDebugCertificate**
Certificate Serial Number
Subject Public Key Information

Certificate extension (X.509 v3):		
Extension Name DebugScenarioEnable	**Criticality** =1	**Value (2bits)** 0b00= No Debug 0b01= Public Debug 0b10= Secure User Debug 0b11= Secure Privilege Debug
Extension Name SoC_ID	**Criticality** =1	**Value (256bits)** SHA256(RootOfTrustPublicKeyHash & AES_{HUK}(Fixed pattern))
Extension Name SPARE	**Criticality** =1	**Value (64bits)** 64bits

The SPARE extension fields of the Primary and Secondary debug certificate are not defined in this document and may contain none security related information. The Primary SPARE extension defines which bits of the Secondary SPARE field are allowed to be set following the same model as the debug scenarios.

Trusted SoC patch certificates

The SoC patch certificates are composed of two certificates; the SoC patch key certificate which contains the Public key (SoCPatchContentCertPK) needed to verify the content certificate that contains the hash of the Application processor (AP) ROM patch and the primary function for patching the SoC such as Cortex-A Cat1 bug correction (may not be security related but for which the correction can be applied only using secure accesses).

The SoC patch key certificate purpose is to optionally transfer the OEM signing authority to a third party company.

Table 8 SoC Patch Key Certificate X.509 v3 Extension fields

Subject Name = **SoCPatchKeyCertificate**		
Certificate Serial Number		
Subject Public Key Information		
Certificate extension (X.509 v3):		
Extension Name FirmwareNVCounter	**Criticality** =1	**Value (5Bits)**
Extension Name SoCPatchContentCertPK	**Criticality** =1	**Value (2048Bits or 256Bits)** (RSA-PSS2048bits / EC-DSA256bits)

It is generally less costly to add on-chip ROM then to add on-chip SRAM on a SoC, therefore it is very common to put a maximum of security primitives in the on-chip ROM. Mechanisms to patch the AP ROM are consequently mandatory.

Table 9 SoC Patch Content Certificate X.509 v3 extension fields

Subject Name = **SoCPatchContentCertificate**
Certificate Serial Number
Subject Public Key Information

Certificate extension (X.509 v3):

Extension Name	**Criticality**	**Value (5Bits)**
FirmwareNVCounter	=1	
Extension Name	**Criticality**	**Value (256Bits)**
APRomPatchHash	=1	SHA256(AP ROM Patch Payload)
Extension Name	**Criticality**	**Value (256Bits)**
SoCPrimaryPatchHash	=1	SHA256(Primary function for patching the SoC)

Trusted SCP firmware certificates

The SCP firmware certificates are composed of two certificates; the SCP firmware key certificate which contains the Public key(SCPFirmwareContentCertPK) needed to verify the SCP Firmware content certificate that contains the hash of the SCP ROM patch and the hash of SCP firmware. The SCP firmware key certificate purpose is to transfer the OEM signing authority optionally to a third party company.

Table 10 SCP Firmware Key Certificate X.509 v3 extension fields

Subject Name = **SCPFirmwareKeyCertificate**

Certificate Serial Number

Subject Public Key Information

Certificate extension (X.509 v3):

Extension Name	**Criticality**	**Value (5Bits)**
FirmwareNVCounter	=1	
Extension Name	**Criticality**	**Value (2048Bits or 256Bits)**
SCPFirmwareContentCertPK	=1	(RSA-PSS2048bits / EC-DSA256bits)

It is generally less costly to add on-chip ROM then to add on-chip SRAM on a SoC, therefore it is very common to put a maximum of security primitives in the on-chip ROM. Mechanisms to patch the AP ROM are consequently mandatory.

Table 11 SCP Firmware Content Certificate X.509 v3 extension fields

Subject Name = **SCPFirmwareContentCertificate**

Certificate Serial Number

Subject Public Key Information

Certificate extension (X.509 v3):

Extension Name	**Criticality**	**Value (5Bits)**
FirmwareNVCounter	=1	
Extension Name	**Criticality**	**Value (256Bits)**
SCPRomPatchHash	=1	SHA256(SCP ROM Patch Payload)
Extension Name	**Criticality**	**Value (256Bits)**
SCPFirmwareHash	=1	SHA256(SCP Firmware)

The System Control and Power sub-system (SCP) is not capable of downloading neither verifying its firmware. The ‘A’ class Cortex processor with the TrustZone technology extension is copying the SCP firmware into the SCP on-chip local SRAM before allowing the SCP to boot.

Trusted OS firmware certificates

The Trusted OS firmware certificates are composed of two certificates; the Trusted OS key certificate which contains the G&D Public key needed to verify the Trusted OS content certificate which contains the hash of the MobiCore firmware.

The Trusted OS firmware key certificate purpose is to transfer the OEM signing authority to Giesecke & Devrient.

Table 12 Trusted OS Firmware Key Certificate X.509 v3 extension fields

Subject Name = **TrustedOSFirmwareKeyCertificate**		
Certificate Serial Number		
Subject Public Key Information		
Certificate extension (X.509 v3):		
Extension Name FirmwareNVCounter	**Criticality** =1	**Value (5Bits)**
Extension Name G&DPK	**Criticality** =1	**Value (2048Bits or 256Bits)** (RSA-PSS2048bits /EC-DSA256bits)

The MobiCore firmware also implements a chain of trust therefore the MobiCore Hash Included in the content certificate do not cover the complete firmware.

Table 13 Trusted OS Firmware Content Certificate X.509 v3 extension fields

Subject Name = **TrustedOSFirmwareContentCertificate**		
Certificate Serial Number		
Subject Public Key Information		
Certificate extension (X.509 v3):		
Extension Name FirmwareNVCounter	**Criticality** =1	**Value (5Bits)**
Extension Name MobiCoreHash	**Criticality** =1	**Value (256Bits)** SHA256(MobiCoreFirmware)

Non-Trusted World UEFI firmware certificate

The UEFI firmware certificates are composed of two certificates; the UEFI firmware key certificate which contains the Public key needed to verify the content certificate that contains the hash of the UEFI firmware.

The UEFI firmware key certificate purpose is to transfer the OEM signing authority optionally to a third party company.

Table 14 UEFI Firmware Key Certificate X.509 meaningful fields

Subject Name = **UEFIFirmwareKeyCertificate**
Certificate Serial Number
Subject Public Key Information

Certificate extension (X.509 v3 only):		
Extension Name RichOSNVCounter	**Criticality** =1	**Value (12Bits)**
Extension Name UEFIFirmwareContentCertPK	**Criticality** =1	**Value (2048Bits or 256Bits)** (RSA-PSS2048bits /EC-DSA256bits)

If UEFIFirmwareContentCertPK or UEFIFirmwareHash is equal to zero, it means that the Rich OS firmware can be freely replaced on the SoC. When done so, the Primary and secondary debug certificate are advised to allow public debug only to guarantee that OEM with TEE enabled on the field are not endangered by cloning.

Such configuration allows enabling on customer configurable open platform the Sepia defined goals of ensuring user privacy and providing security robustness for financial services on open mobile devices and computers.

Table 15 UEFI Firmware Content Certificate X.509 v3 extension

Subject Name = **UEFIFirmwareContentCertificate**		
Certificate Serial Number		
Subject Public Key Information		
Certificate extension (X.509 v3):		
Extension Name RichOSNVCounter	**Criticality** =1	**Value (12Bits)**
Extension Name UEFIFirmwareHash	**Criticality** =1	**Value (256Bits)** SHA256(UEFI firmware)

3.3 SoC mechanism to download TEE firmwares

The TEE firmware images (SoC Patch, SCP firmware and MobiCore) are included in the global SoC firmware image and are downloaded by the Firmware Loader Software using external interfaces such as e.g. USB, Ethernet, Uart during the standard Trusted boot process (not described in this document).

Over The Air (OTA) update is possible as long as the NVM memory that is storing the firmware images is made accessible by the updating software component (trusted or non-trusted). In case of failure to OTA update the firmware images, the global SoC firmware image needs to be re-installed using the standard trusted boot process through external interfaces such as e.g. USB, Ethernet, Uart.

Such processing is made to ensure denial of service on OTA TEE firmware update is prevented as required by Sepia Mobile Platform Attack and Threat Model.

3.4 SoC firmwares anti-replay attacks protections

If a firmware image is updated to fix security vulnerabilities, it must not be possible to roll back the firmware image to a previous unsecure version.

To protect against replay attacks, each signed firmware image must contain a counter value that is checked against a Non-Volatile (NV) counters stored in the SoC.
The counter value stored in the firmware image must be of equal or higher value then the one stored in the SoC.

The NV counters can be implemented using a dedicated SoC hardware resource or additional OTP/Fuse memory.
The Firmware NV counter must have a minimum of 31 states plus an “overflow” state.
The Rich OS NV counter must have a minimum of 4095 states plus an “overflow” state.

The NV firmware counter can also be used to revoke third parties private keys if they are compromised. The revocation is performed by signing a new trusted key certificate with updated keys and with Firmware NV counter incremented.

An additional NV counter is required for the Secure Storage of elements as required by OMTP [3] against replay attacks.

3.5 SoC firmwares anti-cloning protections

Optionally, to protect the SoC firmware against cloning it is possible to bind the SoC firmware certificates and firmwares image uniquely per SoC.

The binding is realized by decrypting first the SoC firmware using an OEM provisioned OTP/Fuse Secret Symmetric Key (SSK) followed by an encryption and the replacement of the SoC firmware using a reproducible secret unique per device symmetric key.

This unique per device symmetric key is created using AES_{HUK}(Fixed pattern) and is also referred as the Binding Secret Symmetric Key (BSSK).

Table 16 Anti-cloning protection OPT/Fuse Data Fields

Name	Size (bits)	Description
Secret Symmetric Key (Optional)	≥128	Key used for firmware image decryption

3.6 SoC trustworthiness proving

In order for the SoC to optionally prove its trustworthiness to the external world and particularly to the OEM and financial institution, an ECC private key can be provisioned into the OTP/Fuse memory storage.

Table 17 Trustworthiness proving OPT/Fuse Data Fields

Name	Size (bits)	Description
Endorsement Key (optional)	≥ 256	ECC private key of 256 Bits

During the manufacturing process, the Endorsement Key is calculated by the OEM secure servers or internally to the SoC and is stored into OTP/Fuse. The corresponding public key is securely sent with the SoC_ID to the OEM database and financial institution for storing. At run-time, each SoC can sign its SoC_ID using its private Endorsement key and send it to the OEM or financial institution to prove its trustworthiness.

At run-time, the public key corresponding to the Endorsement key can be recalculated using a stored G parameter (e.g. ROM).

ECC is advised to be used instead of RSA (2048bits) for OTP/Fuse cost reduction.

3.7 SoC Boot ROM patching

It is generally less costly to add on-chip ROM than to add on-chip SRAM on a SoC, therefore it is very common to put a maximum of security primitives in the on-chip ROM. Mechanisms to patch the SoC ROM are consequently mandatory.
To do so, an area in the Trusted on chip SRAM is reserved to contain a patch table with links for replacement code of on-chip ROM. The on-chip ROM implements software hooks to re-direct the processor program flow to the SRAM patch table in sensible ROM'ed software components. The patch code is written by the owner of the SoC Boot ROM and validated by the OEM. It is OEM choice to either own the SoC Patch Private Key for signing the patch code or to transfer this responsibility to the SoC Boot ROM owner (e.g. ARM).

3.8 SoC Hardware lock mechanism activation

ARM TrustZone technology implements safeguards such that after a proper configuration of the TEE during the Trusted Boot process and after the proper patching of SoC and ROMs hardware locks are configured. The primary lock is the so called CP15Disable which protects the TrustZone technology from any software change until the 'A' Class processor is reset. In this particular case, secure accesses cannot override this protection and the so called CP15Disable particularly affects the Trusted MMU configuration and memory mapping of the TEE, exception vectors, interrupts and cache configuration. Other locking mechanisms are implemented at SoC interconnect and SCP level for the exact same purpose.

4 ARM SoC Software architecture for TEE OS isolation and Apps sandboxing

4.1 TEE OS isolation against trusted user apps on a virtualized ARM CPU

ARM CPU hardware mechanism for trusted kernel isolation from trusted user apps

The 'A' Class processor memory management unit (MMU) translates virtual to physical addresses and also specifies a number of permission attributes for each memory page (4kbytes, 64Kbytes, 1Meg, 16Meg). These permission attributes define the access permission for the memory address space against the type of accesses (Read or Write) and privilege state. The following configuration capability is available in the Trusted World for protecting the Trusted OS kernel from its Trusted User applications.

Table 18 MMU permission per page for protecting Trusted OS from Trusted Apps

Privileged (Trusted OS kernel)	**Unprivileged (trusted User Apps)**	**Description**
No Access	No Access	Permission Fault, access is aborted
Read	No access	Privileged Read Only **(0)**
Read	Read	Read only
Read/Write	No Access	Privileged Access only **(1)**
Read/Write	Read	No User mode write **(2)**
Read/Write	Read/Write	Full Access **(3)**

Consequently, the Trusted OS kernel and its privileged drivers can be made not accessible by Trusted User Application accesses using configuration **(1),** this includes SoC hardware privileged resources access such as part of Trusted SRAM, Display controller interface, Trusted I2C, Trusted RTC, Trusted RNG and Trusted Watch Dog Timer. The shared data buffers between the Trusted OS kernel and a Trusted User Application can be made using configuration **(2)** and **(3).**

It is possible to protect the Trusted OS kernel against certain buffer overflow types of attacks using the eXecute Never (XN) configuration on a per page basis (4kbyte, 64 Kbytes, 1Mbyte and 16Mbytes) in the MMU by setting it for **(2)** and **(3)** configuration, but also more generally for Trusted Os kernel data structure such as stack and heap **(1).** It is also advised to configure the Trusted OS kernel code pages with **(0).**

In the same manner as the NS-bit (AxPROT[1]) is hardware propagated thru CPU L1 and L2 cache. The privilege information is carried inside the CPU and outside of it in interconnect transaction thru the AWPROT[0] and ARPROT[0] signals:

- AWPROT[0]: Write transaction – LOW is unprivileged and HIGH is privileged.

- ARPROT[0]: Read transaction – LOW is unprivileged and HIGH is privileged

However, the L2 cache speculative prefetch and the data write posting from L2 cache level to SoC level are always performed using AxPROT[0]=1 (privilege).
This privilege escalation has no security impact since for data speculative prefetching the MMU is verifying if these data are allowed to be accessed or not by the CPU in the L2 cache. Regarding the write access, before being issued to the L2 cache level, the MMU have already verified the related access permission.
Architecturally speaking, the MMU is disambiguating the privilege and unprivileged attributes and the privilege information is no longer needed after the MMU boundary from the 'A' class processor's point of view.

G&D MobiCore TEE OS isolation architecture on ARM CPU

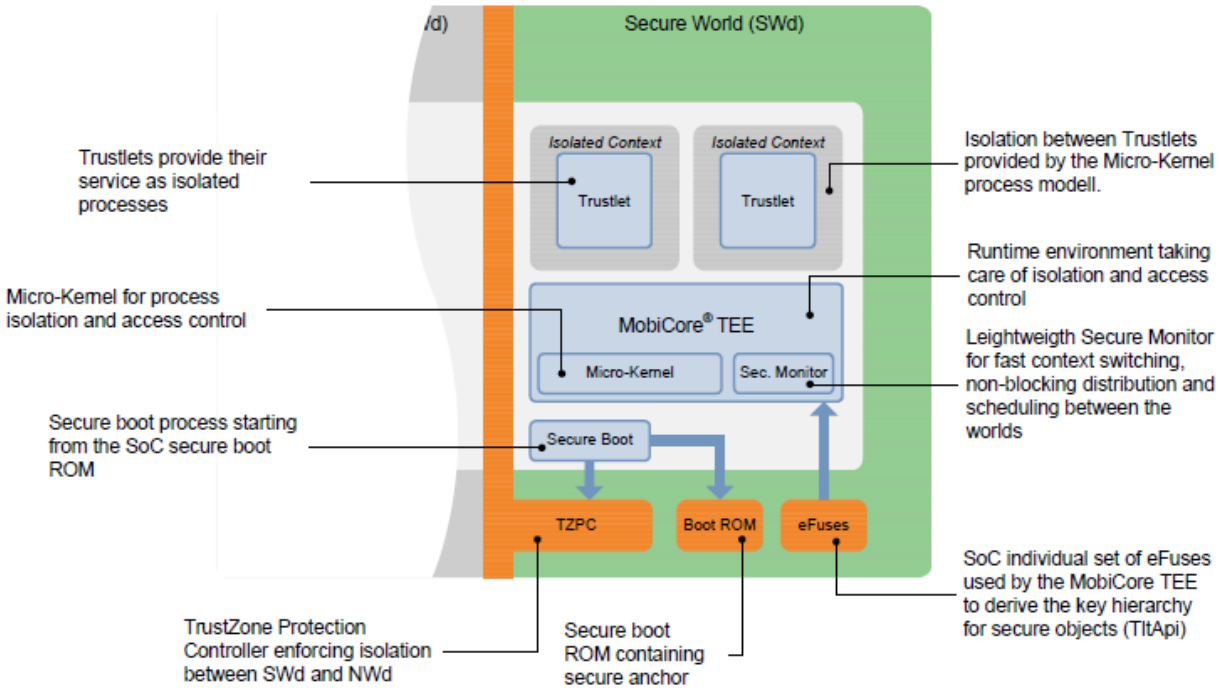

Figure 4: G&D MobiCore TEE OS Components

The MobiCore TEE security is based on a secure boot process with a secure anchor located in immutable on-chip Boot ROM. During the secure boot process MobiCore configures the SoC in a way that software executed in the Non-Trusted World (NWd) cannot access peripherals and data dedicated to the SWd. This is achieved by configuring the SoCs TrustZone Protection Controller (TZPC) before control is handed over to any software component considered to have a lower level of trust as MobiCore.

The MobiCore TEE comprises of the micro-kernel (L4) running in privileged kernel-mode and the Runtime Environment running in unprivileged user-mode.

The MobiCore micro-kernel provides process isolation based on traditional kernel techniques like separated address spaces, pre-emptive multitasking and inter-process communication. The micro-kernel enforces access permission handling. The implementation of the micro-kernel is based on the ARM CPU hardware mechanisms such as the Memory management Unit (MMU) and Memory Protection Unit (MPU).

The MobiCore Runtime Environment implementation is based on the micro-kernel. It is responsible for access control on a logic level. Trusted peripherals and MobiCore runtime have to be protected against illegal access. Therefore MobiCore enforces strict access control mechanisms.

A Trusted Application is executed as unprivileged MobiCore process. Isolation between Trusted Applications and between Trusted Applications and the MobiCore TEE is provided by the micro-kernel process model. Each time a Trusted Application is started, the MobiCore TEE defines the capabilities of that Trusted Application. The MobiCore kernel ensures compliance of the running Trusted Applications with these capabilities.

The capabilities of a MobiCore Trusted App are comprised of:

- Memory access rights
- Execution priority
- Resource usage
- Interprocess communication rights.

The MobiCore kernel's pre-emptive scheduler runs Trusted Applications processes based on execution priorities assigned by the MobiCore TEE. Processes on the same priority level are scheduled following the round-robin algorithm. Each process is executed in a separated address space. These address spaces are separated from each other Trusted Application and the TEE. The MobiCore Runtime Environment itself runs at a higher priority than each Trusted Application.

ARM SoC hardware mechanism for System scale isolation of trusted kernel from trusted user apps

The privilege information is carried outside of the CPU boundary using the AWPROT[0] and ARPROT[0] signals:

- AWPROT[0]: Write transaction – LOW is unprivileged and HIGH is privileged.
- ARPROT[0]: Read transaction – LOW is unprivileged and HIGH is privileged

As discussed in the previous chapter, the 'A' Class processor disambiguates the privilege access permission using its embedded MMU. The SoC interconnect fabric properly routes the AxPROT[0] information from any Trusted initiator (such as Trusted DMA, Trusted Display, SCP) to any Trusted Target and the SoC interconnect is capable of restricting Trusted Peripherals (such as OTP/Efuse keys, Trusted SRAM, Trusted I2C and Trusted RNG) to be accessed only with privilege accesses (AxPROT[0]=1). But there is no guarantee that any Trusted SoC Initiator if controlled by a Trusted User Application could not escalate privilege and be physically capable of accessing the privilege resources (e.g. OTP/Efuse keys). To protect against Trusted User Applications running on 'A' Class processor accessing forbidden privilege resources using as proxy a Trusted OS Initiator without an MMU, the Trusted OS kernel must consider all Trusted SoC Initiators as privileged resources and thus must not be Trusted User accessible and configurable by direct accesses. Consequently any communication from a Trusted User Application with Trusted Initiators must be made using Trusted OS privilege drivers.

This particular security limitation will be subject of further research on a hardware extension for a memory protection unit to be added in Trusted Initiators. The memory

protection unit will work **coherently** with the ‘A’ Class processor MMU such that the SoC Trusted initiators do not increase in complexity for understanding the SoC mapping and associated security dependencies but rather are under the implicit control of the Trusted OS Kernel and relevant security strategy while being directly accessible by the Trusted User Applications.

The SoC trusted initiators such as DMA, Display Controllers, SCP (‘M’ Class processor) will then be able to disambiguate the unprivileged accesses from the privilege one.

This analysis and consequences are also applicable for protecting against certain buffer overflow types of attacks where an ‘M’ Class processor could be forced to execute in a specific memory location containing a so called “shellcode” such as a SoC write data buffer. Sepia research on a hardware extension for a memory protection unit will provide an adequate solution for protecting and limiting effects of such attacks by disambiguating the unprivileged accesses from the privilege one at SoC level.

4.2 Isolation of Trusted User apps from other Trusted user apps scheduled by a Trusted OS on a virtualized TEE

ARM hardware mechanism helping user apps isolation

On ‘A’ Class processors, a number of Trusted User Applications referred as tasks can run concurrently. Each Task has its own unique page tables configuration residing in physical memory. Typically, much of the memory system is organized so that the virtual-to-physical address mapping is fixed, with page table entries that never change. This typically is used to contain operating system code and data and also the page tables used by individual tasks. Whenever an application is started, the operating system will allocate it a set of page table entries which map both the actual code and data used by the application to physical memory. If the application needs to map in code or extra data space (for example through a malloc() call), the kernel can subsequently modify these tables. When a task completes and the application is no longer running, the kernel can remove any associated page table entries and re-use the space for a new application.

In this way, multiple tasks can be resident in physical memory. Upon a task switch, the kernel switches page table entries to page in the next thread to be run. In addition, the dormant tasks are completely protected from the running task. The goal is to ensure that one specific Trusted User Application cannot access memory resources used by another trusted user application thanks to the MMU. This is made possible in the MMU using an Address Space Identifier (ASID).

When the Trusted OS kernel is in privileged mode and describing the page table of a Trusted User Application it has the option to select the nG (not Global) bit field. When the nG bit is set for a particular page, it means that the page is associated with a specific Trusted User Application (ASID defined in CP15) and it means that this mapping is not global and thus not shared with other Trusted User Applications.

Consequently, when the MMU performs a translation, it uses both the virtual address and an ASID value. The ASID is a number assigned by the Trusted OS to each individual Trusted User application or Tasks. This value is in the range 0-255 and the

value for the current task is written in the ASID register (accessed via CP15 c13 using secure privilege accesses). When a page table walk occurs and the TLB is updated and the entry is marked as non-global, the ASID value will be stored in the TLB entry in addition to the normal translation information. Subsequent TLB look-ups will only match on that entry if the current ASID matches with the ASID that is stored in the entry. This means that it is possible to have multiple valid TLB entries for a particular page (marked as non-global), but with different ASID values. This significantly reduces the software overhead of context switches, as it avoids the need to flush the on-chip TLBs while also providing hardware data isolation between Trusted User Applications.
As a matter of fact, since the ASID and MMU pages configuration are only configurable through secure privilege access, Trusted User Applications are hardware Sandboxed.
The ASID forms part of a larger (32-bit) process ID register that can be used in task aware debugging for debugging specific Trusted User Application.

For more details please refer to the following:

- *ARMv7-A Architecture Reference Manual* [2]. This is ARM's Application class Architecture Reference Manual, and includes details on how TrustZone technology is implemented in its architecture.

G&D Sandboxed Trusted User Apps architecture

Native Applications

See also chapter 4.1 for the description of the MobiCore TEE components. Trusted Applications are realized in MobiCore as unprivileged MobiCore processes based on a micro-kernel process model.

When a Trusted Application is started, the MobiCore TEE associates capabilities of that Trusted Application. The capabilities of a MobiCore Trusted App are comprised of:

- Memory access rights
- Execution priority
- Resource usage
- Interprocess communication rights.

Private data of a Trusted Application is not accessible by another Trusted Application. Nevertheless MobiCore TEE is able to set-up shared memory between two or more address spaces. Illegal memory access is intercepted by the MobiCore kernel and reported to the MobiCore TEE.

SoC resources are accessible only through privilege drivers in the Trusted OS (HW abstraction layer). SoC resources are:

- DMA
- OTP/Efuse
- NV counter
- Crypto accelerators (e.g. RNG)
- Timers

Trusted peripherals like Trusted keypad, Trusted display and Trusted I2C bus are exclusively controlled by the MobiCore Runtime Environment. In order to avoid resource conflicts MobiCore applies an access control policy to the usage of such resources. Trusted Applications may access the secure drivers only via the Runtime Environment using a Trusted Application API.

Secure Storage of Trusted Applications Data

MobiCore does not require direct access to secure non-volatile storage. Instead it offers a mechanism to wrap data into Secure Objects, which contain the data in encrypted and integrity protected form. Secure Objects are passed by the Trusted Applications or MobiCore to the Rich OS client application and can be stored in an arbitrary insecure location.

Once that secured data is needed again, its Secure Object must be returned to MobiCore for checking and unwrapping of the data.

Secure Objects can be employed by Trustlets and the MobiCore. In fact, a Trustlet code blob stored in the RichOS for dynamic loading is basically a Secure Object owned by MobiCore itself.

Every Secure Object is encrypted with a key derived from device hardware unique key stored in an OPT/Fuse. For every Trusted Application a unique key is derived. The keys are managed by MobiCore and cannot be exported. This ensures that any wrapped data cannot be unwrapped by any other party besides the platform that has generated it.

Interpreted Apps

In this section, we discuss our approach for hosting managed environments in a TrustZone technology protected environment. We specially focus on small, embedded virtual machines i.e. the Java2 MicroEdition (J2ME) Squak VM [4] and the Microsoft .NET MicroFramework [5]. Both virtual machines provide a set of APIs and runtime-libraries that are offered to applications which are executed in the virtual machine's context.

Our approach relies on two important components: the trusted container (TC) and trusted applets or trustlets (TL).

A trusted container (TC) is basically a runtime environment that provides a specific API and set of runtime libraries to applications (i.e. J2ME API or common language API). Moreover, it protects the integrity and confidentiality of code and data processed inside such a TC. Code that is executed inside the TC cannot be manipulated by applications (including the Rich OS) from the untrusted world. This is also true for applications which are prohibited from accessing other TCs than their own. The code executed in the TC is called trustlet, similar to the nomenclature used for MobiCore applications.

Advantages of using managed environments:
Several advantages when using virtual machines exist, e.g. the managed environment reduces the risk of exploiting implementation bugs caused by faulty implementations. Moreover, the general number of exploitable bugs is reduced as developers do not have to care about memory handling or pointer arithmetic which are a common source of implementation faults. In addition, virtual machines provide means for downloading and authenticating executable code per se. Hence, it is possible to store the code outside of the trusted environment and load it into the trusted execution environment (TEE) when needed.

However, there are some disadvantages when using managed environments:
The use of a VM and its runtime involves higher memory consumption than executing native applications. Moreover, managed apps and interpreted code are typically slower and require Just-in-time compilation (JIT) or dynamic-ahead-of-time compilation (DAO)

to be competitive with native apps. Furthermore, the additional binary code and data required for the JIT or DAO compiler further increases the total memory consumption.

We investigated three approaches to create trusted containers:

- The virtual machine is executed barebone i.e. without operating system on the SoC. In this case, the VM contains a small library that handles memory management and provides I/O functionality. Moreover, the library provides a small storage functionality to allow the application to store data temporarily. In addition, the library provides access to non-volatile memory in order to provide access to cryptographic material. The drawback of this approach is that only one (Java) application can be executed. Hence, the number of provided services is limited to this application.

- The virtual machine is executed in a container provided by the secure world operating system (e.g. L4 compartment, MobiCore etc.). In this case, multiple VM can be executed providing a rich set of services. Moreover, the basic I/O functionality and communication mechanisms to communicate with other trustlets or the untrusted world are provided by the secure world operating system.

- The virtual machine is executed barebone i.e. without operating system and provides isolation between applications via JSR 121 [6] or in case of .NET via application domains. This case is similar to (1.), however, due to the isolation features of JSR 121, the VM is able to execute applications concurrently.

Creation of a trusted container:
The start of a trusted container is triggered in two ways: a TC is either started during system boot when all services running in the Trusted World are started or if an application from the Non-Trusted world requests a specific service inside the Trusted World. In both approaches, the corresponding trustlet is loaded and its integrity and authenticity is verified before execution.

Destruction of a Trusted Container:
In case of design 1, the container is destroyed when the phone is turned off. In case 2, the VM and its TC the compartment manager of the operating systems simply deletes the container and frees all its resources and its internal state. The current state may also be saved and reloaded at a later point in time.

Input/output:
Our design supports two different types of information exchange mechanisms with the Non-Trusted World which are based on register transfer and shared memory. The basic design principles are discussed in [1].

Memory management:
Each TC is responsible for managing its internal state and memory state. In our current design, we statically allocate a fixed set of memory pages to the trusted world when the platform boots. However, the memory requirements of applications running inside TC may vary over time, and possibly even exceed the capacity that was allocated at boot. To support this, in the future we may need a small OS kernel running in the trusted world to enable demand paging to the DDR memory typically found in smartphones. Encryption must be used to ensure that paging does not compromise the integrity and confidentiality benefits that the TC currently provides.

The designs discussed in the previous sections may also be extended for virtual compartments and cloud computing scenarios. Details can be found in [7].

5 Future directions of TEE operating system isolation with HW and SW components

5.1 Privacy for isolated Apps

It has been shown by recent data scandals (e.g. tracking of mobile devices by Google and loss of privacy-relevant data by Sony) that mobile devices are increasingly becoming essential carriers and hence also providers of identities for individuals – in both, the business as well as the private field.

The PrimeLife project (www.primelife.eu) has spent significant resources on researching this domain in detail and G&D drove the Mobile Privacy aspects within the PrimeLife project.

The corresponding publications (see http://www.primelife.eu/results/documents , and especially the publications of Workpackage 6) have drawn attention to the fact that different applications on mobile devices will need to relate to different partial identities of the individual (e.g. a mobile banking application will need to access different data sets of the individual user than a mobile social networking app would / should). Further, the various applications must not have access to the data handled in other mobile applications. Hence, isolation is a must and the protection of the private data for each application has to be an inherent concept of each application.

The PrimeLife demonstrator on Privacy for Mobile Applications has shown this in the exemplary case of an electronic Curriculum Vitae and communication with a headhunter via a Cloud-based service. For this, the PrimeLife project leveraged G&D's secure Micro SD Card technology.

Within SEPIA, the lessons learned in PrimeLife will now be taken further and the conceptual findings for one exemplary application and one exemplary technology of PrimeLife will be translated to the ARM TrustZone technology based platform and the MobiCore technology. Infineons TPM technology may also serve as identity providing Secure Element in the interplay of different technologies and identity providing technologies of the Mobile Device and the backend technologies of the "Internet of the Future".

5.2 Information Flow Security

The next action steps include the investigation of several viable architectures to enforce Information Flow Security on a system wide level. This includes the integration of information tracking engines and a policy enforcement engine on the platform (e.g. in the platforms operating system).

Moreover, we will investigate what implementation level IFS could be best employed to complement security enhanced virtualization. We want to research the applicability of low-level virtual machines for tracking the flow of information through a mobile device for security purposes. Our concept aims at a low-level virtual machine which virtualizes the actual processor to a virtual processor which provides an instruction set amenable to information flow tracking. To minimize the overhead induced by this approach, and thus to conserve power, and also to provide protection against a wide range of possible

attacks, we propose to combine static analysis on the source code with a dynamic run-time monitoring and enforcement of high level policies.

5.3 Hardware extensions and certification

Current ARM ‘A’ Class processors are general purpose, and hence able to execute software to implement all the cryptographic algorithms of interest (such as SHA256 and AES). However, compared to dedicated hardware implementations, the software implementations are typically slower, more power hungry and subject to well known software based attacks such as cache timing.

In Sepia WP2, ARM will conduct research with the goal of adding high performance AES, SHA cryptographic and Polynomial multiplication instructions that will be capable of competing with dedicated crypto accelerators in the specific use case of large files hashing, encrypting and decryption from and to DDR memory and digital signature generation and verification.

As matter of fact, the DDR access latency from the CPU (generally about 100ns) and interconnect bandwidths are the performance bottlenecks in use cases such as cloud computing, VPN and internet-of-things.

This research will aim to show that the high-speed standalone crypto-engines do not provide anymore a performance differentiation compared to ‘A’ Class cryptographic instructions. Such results will allow reducing the cryptographic processing fragmentation in the SoC space and allow PET to be widely integrated.

6 References

[1.] ARM Security Technology - Building a Secure System using TrustZone Technology, 2005, http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf

[2.] ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition (ARM DDI 0406)

[3.] OMTP Advanced trusted environment OMTP TR1, version 1.1, 28 May 2009

[4.] The Squawk Project, Oracle Labs, http://labs.oracle.com/projects/squawk/

[5.] .NET Micro Framework, Microsoft, http://www.microsoft.com/en-us/netmf/default.aspx

[6.] Java Community Process, JSR 121: Application Isolation API Specification, http://www.jcp.org/en/jsr/detail?id=121

[7.] Kurt Dietrich and Johannes Winter, Towards a Trustworthy, Lightweight Cloud Computing Framework for Embedded Systems, Trust 2011, 22-24 June 2011, Pittsburgh, PA USA